**Lab 3 – Database: Create the circle and the Block**

In this lab, we will add a circle to model space, create a block and add a block reference to model space. (It references the block). The focus of this lab should be on the fundamentals of database access in AutoCAD. The major points are Transactions, ObjectIds**,** Symbol Tables such as the BlockTable. Other objects are used in conjunction with our steps such as Point3d and Vector3d.

Open the Lab3 project or continue where you left off in your Lab2 code by copying the steps below. Work through the steps to create a circle, block and block reference. Below these steps is a discussion about transactions, Exception Handling and the using keyword.

Note: Copy steps 1-10 after this “if” statement in the Case Circle from Lab 2.

if ((getRadiusResult.Status == PromptStatus.OK))

// Begining of Lab3. Create the Circle or Block and BlockReference

// 1. Declare a Database variable and instantiate it

// using the Document.Database property of the editor created above. (ed)

// Note: Add the Autodesk.AutoCAD.DatabaseServices; namespace for Database

// and Transaction, use the using keyword (above the class declaration)

// 2. Declare a Transaction variable; instantiate it using the

// TransactionManager.StartTransaction method of the Databse

// created in step 1.

// 3. Add a try, catch and finally block. Move the try closing curly

// brace right after step 8. Put the catch statement after this.

// Enclose step 9 in the catch call. Enclose step 10 in the finally call.

// (Build the project and fix any problems).

// 4. Declare a Circle variable and create it using the new keyword.

// Use the the Value property of the PromptPointResult created in

// Lab2 for the first parameter. For the second parameter (normal) use

// Vector3d.ZAxis. Use the Value property of the PromptDoubleResult

// (created in Lab2) for the radius.

// Note: Need to add Autodesk.AutoCAD.Geometry; namespace for Vector3d.

// 5. Declare a BlockTableRecord variable. Instatiate it using the

// GetObject method of the Transaction variable created in step 2.

// Use the CurrentSpaceId property of the Database variable created in

// step 1 for the first parameter. (ObjectId) For the second parameter

// use OpenMode.ForWrite. We are adding the circle to either ModelSpace

// or PaperSpace. (the CurrentSpaceId determines this)

// 6. Add the Circle to the BlockTableRecord created in step 5. Use the

// AppendEntity method and pass in the circle created in step 4.

// 7. Tell the transaction about the new circle so that it can autoclose

// it. Use the AddNewlyCreatedDBObject method. The first argument is the

// circle. Use True for the second argument.

// 8. Commit the transaction by calling the Commit method. If the code gets

// this far everything should have worked correctly.

// 9. Declare an Exception variable for the Catch.

// (add "(Exception ex)" to the catch keyword)

// Use the WriteMessage of the Editor variable (ed) created in Lab2.

// Use "problem due to " + ex.Message for the Message parameter.

// If an error occurs the details of the problem will be printed

// on the AutoCAD command line.

// 10. Dispose the transaction by calling the Dispose method

// of the Transaction created in step 2. This will be called

//whether an error on not occurred.

Note: Copy steps 11-36 after this “if” statement in the Case Block from Lab 2.

if (blockNameResult.Status == PromptStatus.OK)

// 11. Declare a Database variable and instantiate it using the

// Document.Database property of the editor created above. (ed)

// 12. Declare a Transaction variable; instantiate it using the

// TransactionManager.StartTransaction method.

// 13. Add a try, catch and finally block. Move the closing curly

// brace right after step 34. Put the catch statement after this.

// Enclose step 35 in the catch call. Enclose step 36 in the finally call.

// 14. Declare a BlockTableRecord variable. Create it using the

// new keyword.

// 15. Set the name of the BlockTableRecord. Use the

// StringResult property of the PromptResult variable above.

// (created in Lab2)

// 16. Declare a variable as a BlockTable. Instiate it using the

// GetObject method of the Transaction. Use the BlockTableId property

// of the Database variable created in step 11 for the first parameter.

// Use OpenMode.ForRead for the second parameter. We are opening for

// read to check if a block with the name provided by the user already exists.

// 17. Add an if statement. Test to see if the BlockTable has a block by

// using the Has method of the variable created in step 16. For the string

// Parameter use the StringResult property of the PromptResult variable above.

// created in Lab2. Check to see if it equals False.

// Move the closing curly brace below Step 34.

// 18. The Block with that name does not exist so add it.

// First make the BlockTable open for write. Do this by calling the

// UpgradeOpen() method of the BlockTable. (created in step 16)

// 19. Add the BlockTableRecord created in step 14. Use the Add method

// of the BlockTable and pass in the BlockTableRecord.

// 20. Tell the transaction about the new Block so that it can autoclose

// it. Use the AddNewlyCreatedDBObject method. The first argument is the

// BlockTableRecord. Use true for the second argument.

// 21. In the next two steps you add circles to the BlockTableRecord.

// Declare a variable as a Circle and instantiate it using

// the new Keyword. For the first argument create a new

// Point3d use (0,0,0), for the second argument use Vector3d.ZAxis,

// use 10 for the Radius argument.

// 22. Append the circle to the BlockTableRecord.

// Use the AppendEntity method pass in the circle from step 21

// 23. Now add another circle to the BlockTableRecord

// Declare a variable as a Circle and instantiate it using

// the new Keyword. For the first argument create a new

// Point3d use (20,10,0), for the second arguement use Vector3d.ZAxis,

// use 10 for the Radius argument.

// 24. Append the second circle to the BlockTableRecord.

// Use the AppendEntity method pass in the circle from step 23

// 25. Tell the transaction manager about the new objects so that

// the transaction will autoclose them. Call the AddNewlyCreatedDBObject

// pass in the Circle created in step 21. Do this again for the circle

// Created in step 23. (use true for the second arguement).

// 26. We have created a new block definition (BlockTableRecord).

// Here we will use that Block and add a BlockReference to modelspace.

// First declare a PromptPointOptions and instantiate it with the new

// keyword. For the message parameter use "Pick insertion point of BlockRef : "

// 27. Declare a PromptPointResult variable. Use the GetPoint method of

// the Editor created in Lab2 (ed). Pass in the PromptPointOptions created

// in step 26.

// 28. Create an if statement and test the Status of the PromptPointResult.

// Test if it is not equal to PromptStatus.OK.

//Place the closing curly brace below step 30.

// 29. If we got here then the GetPoint failed. Call the dispose

// method of the Transaction created in step 11.

// 30. return

// 31. Declare a BlockReference variable. Instatiate it with the new keyword

// Use the Value method of the PromptPointResult for the Position argument. (point3d)

// Use the ObjectId property of the BlockTableRecord created in Step 14 for the

// Second parameter.

// 32. Get the current space. (either ModelSpace or PaperSpace).

// Declare a BlockTableRecord variable, instantiate it using the GetObject

// method of the Transaction created in step 12. Use the CurrentSpaceId property

// of the Database created in step 11. Open it for write. (OpenMode.ForWrite)

// 33. Use the AppendEntity method of the BlockTableRecord created in step 32

// and pass in the BlockReference created in step 31.

// 34. Tell the transaction about the new block reference so that the transaction

// can autoclose it. Use the AddNewlyCreatedDBObject of the Transaction created in

// step 12. Pass in the BlockReference. Use true for the second parameter.

// 35. If the code makes it here then all is ok. Commit the transaction by calling

// the Commit method

// 36. Declare an Exception variable for the Catch.

// (add "(Exception ex)" to the catch keyword)

// Use the WriteMessage of the Editor variable (ed) created in Lab2.

// Use "a problem occured because " + ex.Message for the Message parameter.

// If an error occurs the details of the problem will be printed

// on the AutoCAD command line.

// 37. Dispose the transaction by calling the Dispose method

// of the Transaction created in step 12. This will be called

// whether an error on not occurred. This is the end of Lab3.

// Build and debug by loading in AutoCAD. (use NETLOAD) and run

// the addAnEnt command.

**Transactions, Exception Handling and the using keyword**

Notice that the transaction is disposed of in the finally block. The reason we call it in the finally block is that if Dispose() is called on the transaction *before* Commit(), the transaction is aborted. The assumption is made that any error condition that will *throw* before trans.Commit() should abort the transaction (since Commit would have never been called). If Commit() is called before Dispose(), as is the case when nothing is thrown, the transaction changes are committed to the database. The Catch block is useful for notifying the user of a problem.

Note: The structure of the Try-Catch-Finally block in relation to the transaction objects in the .NET API should be of interest to the keen observer. The fact that we are instantiating objects within the Try block, but never explicitly Dispose() of them, even when an exception occurs may seem troubling, especially if the observer notes that we are actually wrapping unmanaged objects! Remember, however that the garbagecollection mechanism will take care of our memory allocation when resources become strained. This mechanism in-turn calls Dispose() on the wrapper, deleting our unmanaged object under the hood.

It is important to note here that Dispose() behaves differently with the wrapped unmanaged object depending on whether the object is database-resident or not. Dispose() called on a non-database resident object will call delete on the unmanaged object, while Dispose() called on a database-resident object will simply call close().

**Using Keyword**

Beginning with Visual Studio 2005, Visual Basic includes the ***Using*** keyword which wraps an object implementing IDisposable for automatic disposal. Objects which you would normally call ‘dispose’ on can be automatically handled with this keyword. Using the ‘Using’ keyword with transactions then makes a tremendous amount of sense, as it makes your code much more compact.